Data refinement for true concurrency

Brijesh Dongol John Derrick
Department of Computer Science,
The University of Sheffield S1 4DP, UK
B.Dongol@sheffield.ac.uk, J.Derrick@dcs.shef.ac.uk

The majority of modern systems exhibit sophisticated concurrent behaviour, where several system components modify and observe the system state with fine-grained atomicity. Many systems (e.g., multi-core processors, real-time controllers) also exhibit truly concurrent behaviour, where multiple events can occur simultaneously. This paper presents data refinement defined in terms of an interval-based framework, which includes high-level operators that capture non-deterministic expression evaluation. By modifying the type of an interval, our theory may be specialised to cover data refinement of both discrete and continuous systems. We present an interval-based encoding of forward simulation, then prove that our forward simulation rule is sound with respect to our data refinement definition. A number of rules for decomposing forward simulation proofs over both sequential and parallel composition are developed.

1 Introduction

Data refinement allows one to develop systems in a stepwise manner, enabling an abstract system to be replaced with a more concrete implementation by guaranteeing that every observable behaviour of the concrete system is a possible observable behaviour of the abstract. A benefit of such developments is the ability to reason at a level of abstraction suitable for the current stage of development, and the ability to introduce additional detail to a system via correctness-preserving transformations. During development, a concrete system's internal representation of data often differs from the abstract data representation, requiring the use of a *refinement relation* to link the concrete and abstract states.

Over the years, numerous techniques for verifying data refinement techniques have been developed for a number of application domains [30], including methods for refinement of concurrent [10] and real-time [23] systems. However, these theories are rooted in traditional notions of data refinement, where refinement relations are between concrete and abstract states. In the presence of fine-grained atomicity and truly concurrent behaviour (e.g., multi-core computing, real-time controllers), proofs of refinement are limited by the information available within a single state, and hence, reasoning can often be more difficult than necessary. Furthermore, the behaviours of corresponding concrete and abstract steps may not always match, and hence, reasoning can sometimes be unintuitive, e.g., for the state-based data refinement in Section 2, a concrete step that loads a variable corresponds to an abstract step that evaluates a guard.

When reasoning about concurrent and real-time systems, one is often required to refer to a system's evolution over time as opposed to its current state at a single point in time. This paper therefore presents a method for verifying data refinement using a framework that allows one to consider the intervals within which systems execute [16, 18, 29, 32]. Thus, instead of reasoning over the pre and post states of each component, one is able to reason about the component's behaviour over an interval, which may comprise several atomic steps. The concurrent execution of two or more processes is defined as the conjunction of the behaviour of each process in the same interval [1, 26]; hence, reasoning about a component

John Derrick, Eerke Boiten, Steve Reeves (Eds.):
Refinement Workshop 2013
EPTCS 115, 2013, pp. 15–35, doi:10.4204/EPTCS.115.2

AInit: $\neg grd$

Process *ap*	Process *aq*
ap_1: **if** *grd* **then**	aq_1: **if** *b* **then**
ap_2: $m := 1$	aq_2: $grd := true$
ap_3: **else** $m := 2$ **fi**	aq_3: **else skip fi**

Figure 1: Abstract program with guard *grd*

CInit: $v \leq u < \infty$

Process *cp*	Process *cq*
cp_1: **if** $u < v$ **then**	cq_1: **if** $0 < u$ **then**
cp_2: $m := 1$	cq_2: $v := \infty$
cp_3: **else** $m := 2$ **fi**	cq_3: **else** $v := -\infty$ **fi**

Figure 2: Concrete program with guard $u < v$

naturally takes into account the behaviour of the component's environment (e.g., other concurrently executing processes). Using an interval-based framework enables us to incorporate methods for apparent states evaluation [16, 18, 25], which allows one to take into account the low-level non-determinism of expression evaluation at a high level of abstraction.

The main contribution of this paper is an interval-based method for verifying data refinement, simplifying data refinement proofs in the presence of true concurrency. A forward simulation rule for interval-based refinement is developed, and several methods of decomposing proof obligations are presented, including mixed-mode refinement, which enables one to establish different refinement relations over disjoint parts of the state space. We present our theory at the semantic level of interval predicates, i.e., without consideration of any particular programming framework. Hence, the theory can be applied to any existing framework such as action systems, Z, etc. by mapping the syntactic constructs to our interval predicate semantics. The aim of our work is to reason about programs with fine-grained atomicity and real-time properties, as opposed to programs written in, say, Java that allows specification of coarse-grained atomicity using `synchronized` blocks.

Background material for the paper is presented in Sections 2 and 3, clarifying our notions of state-based refinement and interval-based reasoning. Our interval-based refinement theory is presented in Section 4, which includes a notion of forward simulation with respect to intervals and methods for proof decomposition. Methods for reasoning about fine-grained concurrency and a proof of our running example is presented in Section 5.

2 State-based data refinement

Consider the abstract program in Figure 1, written in the style of Feijen and van Gasteren [22], which consists of variables $grd, b \in \mathbb{B}$, $m \in \mathbb{N}$, initialisation *AInit* and processes *ap* and *aq*. Process *ap* is a sequential program with labels ap_1, ap_2, and ap_3 that tests whether *grd* holds (atomically), then executes $m := 1$ if *grd* evaluates to *true* and executes $m := 2$ otherwise. Process *aq* is similar. The program executes by initialising as specified by *AInit*, and then executing *ap* and *aq* concurrently by interleaving their atomic statements.

An initialisation may be modelled by a relation, and each label corresponds to an atomic statement, whose behaviour may also be modelled by a relation. Thus, a program generates a set of *traces*, each of which is a sequence of states (starting with index 0). Program counters for each process are assumed to be implicitly included in each state to formalise the control flow of a program [14], e.g., the program in Figure 1 uses two program counters pc_{ap} and pc_{aq}, where $pc_{ap} = ap_1$ is assumed to hold whenever control of process *ap* is at ap_1. After execution of ap_1, the value of pc_{ap} is updated so that either $pc_{ap} = ap_2$ or $pc_{ap} = ap_3$ holds, depending on the outcome of the evaluation of *grd*.

One may characterise traces using an *execution*, which is a sequence of labels starting with initiali-

sation. For example, a possible execution of the program in Figure 1 is

$$\langle AInit, ap_1, aq_1, aq_2, ap_3 \rangle \tag{1}$$

Using '.' for function application, an execution ex corresponds to a trace tr iff for each $i \in \mathrm{dom}.ex$, $(tr.i, tr.(i+1)) \in ex.i$ and either $\mathrm{dom}.tr = \mathrm{dom}.ex = \mathbb{N}$ or $size.(\mathrm{dom}.ex) < size.(\mathrm{dom}.tr)$. An execution ex is *valid* iff $\mathrm{dom}.ex \neq \varnothing$, $ex.0$ is an initialisation, and ex corresponds to at least one trace, e.g., (1) above is valid. Not every execution is valid, e.g., $\langle AInit, ap_1, ap_2 \rangle$ is invalid because execution of ap_1 after *AInit* causes *grd* to evaluate to *false* and pc_{ap} to be updated to ap_3, and hence, statement ap_2 cannot be executed. Note that valid executions may not be complete; an extreme example is $\langle AInit, AFin \rangle$, where the execution is finalised immediately after initialisation.

Now consider the more concrete program in Figure 2 that replaces *grd* by $u < v$ and b by $0 < u$. Note that u and v are fresh with respect the program in Figure 1. Initially, $v \leq u < \infty$ holds. Furthermore, *cq* (modelling the concrete environment of *cp*) sets v to ∞ if u is positive and to $-\infty$ otherwise. One may be interested in knowing whether the program in Figure 2 *data refines* the program in Figure 1, which defines conditions for the program in Figure 1 to be substituted by the program in Figure 2 [30]. This is possible if every execution of the program in Figure 2 has a corresponding execution of the program in Figure 1, e.g., concrete execution $\langle CInit, cp_1, cq_1, cq_2, cp_3 \rangle$ has a corresponding abstract execution (1).

In general, representation of data within a concrete program differs from the representation in the abstract, and hence, one must distinguish between the disjoint sets of *observable* and *representation* variables, which respectively denote variables that can and cannot be observed. For example, *grd* in Figure 1 and u, v in Figure 2 cannot both be observable because the types of these variables are different in the two programs. To verify data refinement, the abstract and concrete programs may therefore also be associated with *finalisations*, which are relations between a representation and an observable state. Different choices for the finalisation allow different parts of the program to become observable and affect the type of refinement that is captured by data refinement [10, 11, 12]. For the programs in Figures 1 and 2, we assume finalisations make variable m observable. Hence, Figure 1 is data refined by Figure 2 if *ap* is able to execute ap_2 (and ap_3) whenever *cp* is able to execute cp_2 (and cp_3, respectively). We define a *finalised execution* of a program to be a valid execution concatenated with the finalisation of the program, e.g., $\langle AInit, ap_1, aq_1, aq_2, ap_3, AFin \rangle$ is a finalised execution of the program in Figure 1 generated from the valid execution (1). Valid executions are not necessarily complete, and hence, one may observe the state in the "middle" of a program's execution.

To define data refinement, we assume that an initialisation is a relation from an observable state to a representation state, each label corresponds to a statement that is modelled by a relation between two representation states, and a finalisation is a relation from a representation state to an observable state. Assuming '$\fatsemi$' denotes relational composition and *id* is the identity relation, we define the *composition* of a sequence of relations R as

$$comp.R \mathrel{\widehat{=}} \mathsf{if}\ R = \langle\rangle\ \mathsf{then}\ id\ \mathsf{else}\ head.R \fatsemi comp.(tail.R)$$

which composes the relations of R in order. We also define a function *rel*, which replaces each label in an execution by the relation corresponding to the statement of that label.

We allow finite stuttering in the concrete program, and hence, there may not be a one-to-one correspondence between concrete and abstract executions. Stuttering is reflected in an abstract execution by allowing a finite number of labels '*Id*' to be interleaved with each finalised execution of the abstract program, where *Id* is assumed to be different from all other labels, and the relation corresponding to label *Id* is always *id*. Data refinement is therefore defined with respect to a *correspondence function* that maps

concrete labels to abstract labels. A correspondence function is valid iff it maps concrete initialisation to abstract initialisation, concrete finalisation to abstract finalisation, each label of a non-stuttering concrete statement to a corresponding abstract statement, and each label of stuttering concrete statement to *Id*. For the rest of the paper we assume that the correspondence functions under consideration are valid. A program C is a *data refinement* of a program A with respect to correspondence function f iff for every finalised execution exc of C, $exa \mathrel{\widehat{=}} \lambda\, i{:}\mathrm{dom}.exc \bullet f.(exc.i)$ is a finalised execution of A (with possibly finite stuttering) and $comp.(rel.exc) \subseteq comp.(rel.exa)$ holds.

Proving data refinement directly from its formal definition is infeasible. Instead, one proves data refinement by verifying *simulation* between an abstract and concrete system, which requires the use of *refinement relation* to link the internal representations of the abstract and concrete programs. We assume that a relation $r \in X \leftrightarrow Y$ is characterised by a function $fr \in X \to Y \to \mathbb{B}$ where $(x,y) \in r$ iff $fr.x.y.$ holds. As depicted in Figure 3, a refinement relation ref is a *forward simulation* between a concrete and abstract system if:

1. whenever the concrete system can be initialised from an observable state ρ to obtain a concrete representation state τ_0, it must be possible to initialise the abstract system from ρ to result in abstract representation state σ_0 such that $ref.\sigma_0.\tau_0$ holds,
2. for every non-stuttering concrete statement cs, abstract state σ and concrete state τ, if $ref.\sigma.\tau$ holds and cs relates τ to τ', then there exists an abstract state σ' such that the abstract statement that corresponds to cs relates σ to σ' and $ref.\sigma'.\tau'$ holds,
3. for every stuttering concrete statement starting from state τ and ending in state τ', $ref.\sigma.\tau'$ holds whenever $ref.\sigma.\tau$ holds,
4. finalising any abstract state σ (using the abstract system's finalisation) and concrete state τ (using the concrete system's finalisation) results in the same observable state whenever $ref.\sigma.\tau$ holds.

For models of computation that assume instantaneous guard evaluation [25], establishing a data refinement between the programs in Figures 1 and 2 with respect to a correspondence function that maps cp_i to ap_i and cq_i to aq_i for $i \in \{1,2,3\}$ is straightforward. In particular, it is possible to prove forward simulation using *pcuv* below as the refinement relation, where σ and τ are abstract and concrete states, respectively.

$$uv.\sigma.\tau \mathrel{\widehat{=}} (\sigma.grd = (\tau.u < \tau.v)) \wedge (\sigma.b = (0 < \tau.u)) \wedge (\sigma.m = \tau.m)$$
$$pcuv.\sigma.\tau \mathrel{\widehat{=}} uv.\sigma.\tau \wedge \forall i{:}\{1,2,3\} \bullet (\sigma.pc_{ap} = ap_i \Rightarrow \tau.pc_{cp} = cp_i) \wedge (\sigma.pc_{aq} = aq_i \Rightarrow \tau.pc_{cq} = cq_i)$$

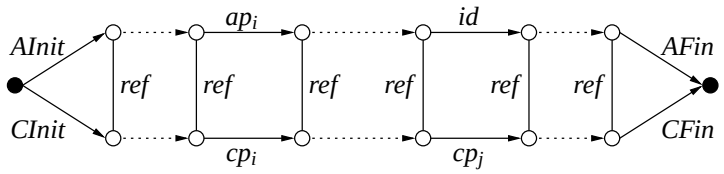


Figure 3: Data refinement via simulation

In a setting with fine-grained atomicity, the program in Figure 2 may be difficult to implement because the guard at cp_1 (which refers to multiple shared variables) is assumed to be evaluated atomically. In reality, there may be interference from other processes while an expression is being evaluated [25]. Furthermore, the order in which variables are read within an expression is often not fixed. To take these

CInit: $v \leq u < \infty$

Process *cp*	Process *cq*
$cp_{1.1}$: $\begin{pmatrix} k_u := u; \\ k_v := v \end{pmatrix}$	cq_1: **if** $0 < u$ **then**
$cp_{1.2}$: $\sqcap$	cq_2: $v := \infty$
$cp_{1.3}$: $\begin{pmatrix} k_v := v; \\ k_u := u \end{pmatrix}$;	cq_3: **else** $v := -\infty$ **fi**
$cp_{1.4}$:	
$cp_{1.5}$: **if** $k_u < k_v$ **then** ...	

Figure 4: Making the atomicity of expression evaluation in Figure 2 explicit

Concrete label	Abstract label
CInit	*AInit*
$cp_{1.1}, cp_{1.3}, cp_{1.5}$	*Id*
$cp_{1.2}, cp_{1.4}$	ap_1
cp_i for $i \in \{2,3\}$	ap_i
cq_i for $i \in \{1,2,3\}$	aq_i
CFin	*AFin*

Figure 5: Correspondence function for data refinement between Figure 4 and Figure 1

circumstances into account, we must consider the program in Figure 4, which splits the guard evaluation at cp_1 in Figure 2 into a number of smaller atomic statements using fresh variables k_u and k_v that are local to process *cp*. Via a non-deterministic choice '$\sqcap$', process *cp* chooses between executions $cp_{1.1}$; $cp_{1.2}$ and $cp_{1.3}$; $cp_{1.4}$, which read the (global values) u and v into local variables k_u and k_v, respectively, in two atomic steps. Evaluation of guard $u < v$ at cp_1 in Figure 2 is then replaced by evaluation of $k_u < k_v$.

A proof of data refinement between the programs in Figures 1 and 2 using forward simulation with respect to *uv* is now more difficult because an (atomic) instantaneous evaluation of *grd* has been split into several atomic statements. A data refinement with respect to a naive correspondence function that matches cp_i for $i \in \{1.1, 1.2, 1.3, 1.4\}$ with *Id*, $cp_{1.5}$ with ap_1, and cq_i with aq_i for $i \in \{1,2\}$ cannot be verified using forward simulation. Instead, one must use the correspondence function in Figure 5. Note that this correspondence function is not intuitive because, for example, execution of $cp_{1.4}$ (which reads u) is matched with execution of ap_1 (which tests *grd*), but is necessary because execution of $cp_{1.4}$ determines the outcome of the future evaluation of the guard at $cp_{1.5}$. The refinement relation used to prove forward simulation is more complicated than *pcuv* (details are elided, but the relation can be constructed using the correspondence function in Figure 5).

Such difficulties in verifying a relatively trivial modification expose the complexities in stepwise refinement of concurrent programs. Further issues arise in the context of real-time properties e.g., transient properties cannot be properly addressed by an inherent interleaving model [17, 18].

This paper presents an interval-based semantics for the systems under consideration, an interval-based interpretation of data refinement in the framework, and a rule akin to forward simulation for proving data refinement. We believe that these theories alleviate many of these issues in state-based reasoning, requiring less creativity on the part of the verifier. For example, the correspondence function always maps each concrete process to an abstract process. By reasoning about the traces of a system over an interval, we are able to capture the effect of a number of atomic statements and interference from the environment at a high-level of abstraction. Unlike the state-based approach described above, which only captures interleaved concurrency, interval-based approaches also allow one to model truly concurrent behaviour. By modifying the type of an interval, one can take both discrete and continuous system behaviours into account.

3 Interval-based reasoning

Our generic theory of refinement is based on interval predicates, generalising frameworks that model programs as relations between pre/post states. We have applied our interval-based methodology to reason

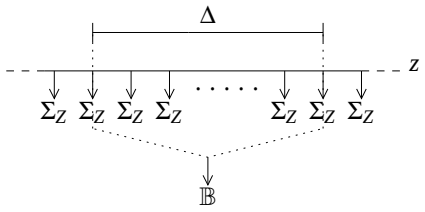


Figure 6: Interval predicate visualisation

about both concurrent [15, 16] and real-time programs [18, 21].

An *interval* in an ordered set $\Phi \subseteq \mathbb{R}$ is a contiguous subset of Φ, i.e., the set of all intervals of Φ is given by:

$$Intv_\Phi \mathrel{\widehat{=}} \{\Delta \subseteq \Phi \mid \forall t, t' : \Delta \bullet \forall t'' : \Phi \bullet t \leq t'' \leq t' \Rightarrow t'' \in \Delta\}$$

We assume the existence of elements $-\infty, \infty \notin \Phi$ such that $-\infty < t < \infty$ for each $t \in \Phi$. $Intv_\Phi$ may be used to model both discrete (e.g., by picking $\Phi = \mathbb{Z}$) and continuous (by picking $\Phi = \mathbb{R}$) systems.

We define the following predicates, which may be used to identify empty intervals, and intervals with a finite and infinite upper bound.

$$\mathsf{empty}.\Delta \mathrel{\widehat{=}} \Delta = \varnothing \qquad \mathsf{finite}.\Delta \mathrel{\widehat{=}} \mathsf{empty}.\Delta \vee (\exists t : \Delta \bullet \forall t' : \Delta \bullet t' \leq t) \qquad \mathsf{infinite}.\Delta \mathrel{\widehat{=}} \neg \mathsf{finite}.\Delta$$

One must often reason about two *adjoining* intervals, i.e., intervals that immediately precede/follow another. For $\Delta_1, \Delta_2 \in Intv_\Phi$, we define

$$\Delta_1 \propto \Delta_2 \quad \widehat{=} \quad (\forall t_1 : \Delta_1, t_2 : \Delta_2 \bullet t_1 < t_2) \wedge (\Delta_1 \cup \Delta_2 \in Intv_\Phi)$$

Thus, $\Delta_1 \propto \Delta_2$ holds iff Δ_2 follows Δ_1 and the union of Δ_1 and Δ_2 forms an interval (i.e., Δ_1 and Δ_2 are contiguous across their boundary). Note that adjoining intervals are disjoint and that both $\Delta \propto \varnothing$ and $\varnothing \propto \Delta$ hold trivially for any interval Δ.

A *state* over $V \subseteq Var$ is of type $State_V \mathrel{\widehat{=}} V \to Val$, where *Var* is the type of a variable and *Val* is the generic type of a value. A *state predicate* is of type $StatePred_V \mathrel{\widehat{=}} State_V \to \mathbb{B}$. A *stream* of behaviours over $State_V$ is given by the function $Stream_{\Phi,V} \mathrel{\widehat{=}} \Phi \to State_V$, which maps each element of Φ to a state over V. To facilitate reasoning about specific parts of a stream, we use *interval predicates*, which have type $IntvPred_{\Phi,V} \mathrel{\widehat{=}} Intv_\Phi \to Stream_{\Phi,V} \to \mathbb{B}$. A visualisation of an interval predicate over $Z \subseteq Var$ is given in Figure 6. The stream $z \in Stream_{\Phi,Z}$ maps each time to a state over Z and the interval predicate depicted in the figure maps Δ and z to a boolean.

We assume pointwise lifting of operators on stream and interval predicates in the normal manner, e.g., if g_1 and g_2 are interval predicates, Δ is an interval and s is a stream, we have $(g_1 \wedge g_2).\Delta.s = (g_1.\Delta.s \wedge g_2.\Delta.s)$. The *chop* operator ';' is a basic operator on two interval predicates [16, 18, 29, 32], where $(g_1 \,;\, g_2).\Delta$ holds iff either interval Δ may be split into two parts so that g_1 holds in the first and g_2 holds in the second, or the upper bound of Δ is ∞ and g_1 holds in Δ. Thus, for a stream s, we define:

$$\begin{array}{rcl} (g_1 \,;\, g_2).\Delta.s & \widehat{=} & \big(\exists \Delta_1, \Delta_2 : Intv_\Phi \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge g_1.\Delta_1.s \wedge g_2.\Delta_2.s\big) \vee \\ & & (\mathsf{infinite}.\Delta \wedge g_1.\Delta.s) \end{array}$$

Note that Δ_1 may be empty, in which case $\Delta_2 = \Delta$, and similarly Δ_2 may empty, in which case $\Delta_1 = \Delta$, i.e., both $(\mathsf{empty} \,;\, g) = g$ and $g = (g \,;\, \mathsf{empty})$ trivially hold, where $\mathsf{empty}.\Delta.s \mathrel{\widehat{=}} (\Delta = \varnothing)$ for all streams

s. Furthermore, in the definition of chop, we allow the second disjunct $\mathsf{infinite}.\Delta \wedge g_1.\Delta$ to enable g_1 to model an infinite (divergent or non-terminating) program.

To model looping of a behaviour modelled by interval predicate g, we use an iteration operator 'g^ω', which is defined as the greatest fixed point of $\lambda\, h \bullet g\,;\, h \vee \mathsf{empty}$. Interval predicates are assumed to be ordered using implication '$\Rightarrow$' and the greatest fixed point allows g^ω to model both finite (including 0) and infinite iteration [21].

$$g^\omega \;\widehat{=}\; \nu z \bullet (g\,;\, z) \vee \mathsf{empty}$$

We say that g *splits* iff $g \Rightarrow (g\,;\, g)$ and g *joins* iff $(g\,;\, g^\omega) \Rightarrow g$. If g splits, then whenever g holds in an interval Δ, g also holds in any subinterval of Δ. If g joins, then g holds in Δ whenever there is a partition of Δ such that g holds in each interval of the partition. Note that if g splits, then $g \Rightarrow g^\omega$ [21]. Splits and joins properties are useful for decomposing proof obligations, for instance, both of the following hold.

$$(g \Rightarrow g_1) \wedge (g \Rightarrow g_2) \;\Rightarrow\; (g \Rightarrow g_1\,;\, g_2) \qquad \text{provided } g \text{ splits} \tag{2}$$

$$(g \wedge g_1)\,;\, (g \wedge g_2) \;\Rightarrow\; g \wedge (g_1\,;\, g_2) \qquad \text{provided } g \text{ joins} \tag{3}$$

One must often state that a property only holds for a non-empty interval, and that a property holds for an immediately preceding interval. To this end, we define:

$$\underline{g} \;\widehat{=}\; g \wedge \neg\mathsf{empty} \qquad\qquad \ominus g.\Delta.s \;\widehat{=}\; \exists \Delta_0 : \mathit{Intv}_\Phi \bullet \Delta_0 \propto \Delta \wedge g.\Delta_0.s$$

Note that if g holds in an empty interval, then $\ominus g$ trivially holds. Also note how interval predicates allow the behaviour outside the given interval to be stated in a straightforward manner because a stream encapsulates the entire behaviour of a system. We define the following operators to formalise properties over an interval using a state predicate c over an interval Δ in stream s.

$$\boxdot c.\Delta.s \;\widehat{=}\; \forall t : \Delta \bullet c.(s.t) \qquad\qquad \diamondsuit c.\Delta.s \;\widehat{=}\; \exists t : \Delta \bullet c.(s.t)$$

That is $\boxdot c.\Delta.s$ holds iff c holds for each state $s.t$ where $t \in \Delta$ and $\diamondsuit c.\Delta.s$ holds iff c holds in some state $s.t$ where $t \in \Delta$. Note that $\boxdot c$ trivially holds for an empty interval, but $\diamondsuit c$ does not. For the rest of this paper, we assume that the underlying type of the interval under consideration is fixed. Hence, to reduce notational complexity, we omit Φ whenever possible.

Example 1. We present the interval-based semantics of the programs in Figures 1 and 2. Interval-based methods allow one to model true concurrency by defining the behaviour of a parallel composition $p \| q$ over an interval Δ as the conjunction of the behaviours of both p and q over Δ (see [15, 16, 18] for more details). Others have also treated parallel composition as conjunction, but in an interleaving framework with predicates over states as opposed to intervals (e.g., [1, 26]). Sequential composition is formalised using the chop operator. We assume $[grd]$ denotes an interval predicate that formalises evaluation of grd. Details of guard evaluation are given in Section 5.1. The interval-based semantics of the programs in Figures 1 and 2 are respectively formalised by the interval predicates (4), (5), (6) and (7) below. Assuming that ρ is an observable state, conditions (4) and (5) formalise the behaviours of $AInit.\rho$ and $CInit.\rho$, respectively. Assuming that ρ has an observable variable M that is represented internally by m, and that σ and τ are abstract and concrete states, respectively, the behaviours of both $AFin.\sigma.\rho$ and $CFin.\sigma.\rho$ are formalised by (8) and (9), respectively. We assume ';' binds more tightly than binary boolean operators.

$$\underline{\boxdot \neg grd} \tag{4}$$

$$\underline{\boxdot(v \leq u < \infty)} \tag{5}$$

$$\overbrace{([grd]\,;\ \underline{\boxdot(m=1)} \vee [\neg grd]\,;\ \underline{\boxdot(m=2)})}^{\text{Process } ap} \wedge \overbrace{([b]\,;\ \underline{\boxdot grd} \vee [\neg b])}^{\text{Process } aq} \tag{6}$$

$$\overbrace{([u<v]\,;\ \underline{\boxdot(m=1)} \vee [u \geq v]\,;\ \underline{\boxdot(m=2)})}^{\text{Process } cp} \wedge \overbrace{([0<u]\,;\ \underline{\boxdot(v=\infty)} \vee [0 \geq u]\,;\ \underline{\boxdot(v=-\infty)})}^{\text{Process } cq} \tag{7}$$

$$\sigma.m = \rho.M \tag{8}$$

$$\tau.m = \rho.M \tag{9}$$

By (4), *AInit* returns an interval predicate $\underline{\boxdot \neg grd}$, which states that $\neg grd$ holds throughout the given interval, and the interval is non-empty. Condition (5) is similar. Condition (6) models the concurrent behaviour of processes *ap* and *aq*. Process *ap* either behaves as $[grd]\,;\ \underline{\boxdot(m=1)}$ (grd evaluates to true, then the behaviour of $m := 1$ holds) or $[\neg grd]\,;\ \underline{\boxdot(m=2)}$ ($\neg grd$ evaluates to true, then the behaviour of $m := 2$ holds, i.e., the interval under consideration is non-empty and $m = 2$ holds throughout the interval). Process *aq* is similar, but also models the assignments to grd.

Note that the points at which the intervals are chopped within (6) and (7) are unsynchronised. For example, suppose process *ap* behaves as $[grd]\,;\ \underline{\boxdot(m=1)}$ and *aq* behaves as $[b]\,;\ \underline{\boxdot grd}$ within interval Δ of stream y, i.e,. $([grd]\,;\ \underline{\boxdot(m=1)} \wedge [b]\,;\ \underline{\boxdot grd}).\Delta.y$ holds for some interval Δ and abstract stream y. By pointwise lifting, this is equivalent to $([grd]\,;\ \underline{\boxdot(m=1)}).\Delta.y \wedge ([b]\,;\ \underline{\boxdot grd}).\Delta.y$. The two processes may now choose to split Δ independently. This includes the possibility of Δ being split at the same point, which occurs if both guard evaluations are completed at the same time.

4 A general theory of refinement

We aim to verify data refinement between systems whose behaviours are formalised by interval predicates. Hence, we present interval-based data refinement (Section 4.1) and define interval-based refinement relations (Section 4.2), enabling formalisation of refinement relations in an interval-based setting. Section 4.3 presents our generalised proof method, which is inspired by state-based forward simulation techniques. Section 4.4 presents a number of decomposition techniques for forward simulation.

4.1 Data refinement

Existing frameworks for data refinement model concurrency as an interleaving of the atomic system operations [2, 9, 10, 30]. This allows one to define a system's execution using its set of operations. The traces of a system after initialisation are generated by repeatedly picking an enabled operation from the set non-deterministically then executing the operation. Such execution models turn out to be inadequate for reasoning about truly concurrent behaviour, e.g., about *transient* properties in the context of real-time systems [18]. The methodology in this paper aims to allow modelling of truly concurrent system behaviour. Each operation is associated with exactly one of the system processes and execution of a system (after initialisation) over an interval Δ is modelled by the conjunction of the behaviours of each operation over Δ (see Example 1). It is possible to obtain interleaved concurrency from our truly concurrent framework via the inclusion of permissions [6, 16].

Action refinement for true concurrency in a causal setting is studied in [28], and a modal logic for reasoning about true concurrency is given in [4]. Frameworks for concurrent refinement in real-time contexts have also been proposed (e.g., [24, 31]). We are however not aware of a method that allows data refinement under true concurrency.

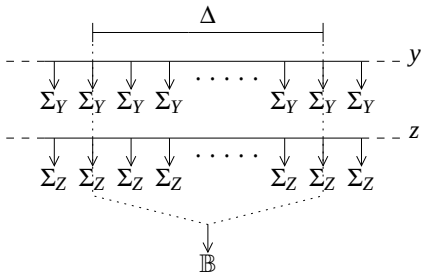


Figure 7: Interval relation visualisation

We let *Proc* denote the set of all process identifiers. For $P \subseteq Proc$ and $N, Z \subseteq Var$, respectively denoting the sets of observable and representation variables, a *system* is defined by a tuple:

$$C \;\widehat{=}\; (CI, (COp_p)_{p:P}, CF)_{N,Z}$$

where $CI : State_N \to IntvPred_Z$ models the initialisation, $COp_p \in IntvPred_Z$ for each $p \in P$ model the system processes, and $CF : State_Z \to State_N \to \mathbb{B}$ denotes system finalisation. The set of observable states at the start and end of an execution of system C is given by:

$$obs_N.C \;\widehat{=}\; \left\{ (\rho, \rho') : State_N \times State_N \;\middle|\; \begin{array}{l} \exists \Delta : Intv, z : Stream_Z \bullet \\ (\ominus CI.\rho \wedge \bigwedge_{p:P} COp_p).\Delta.z \wedge \exists t : \Delta \bullet CF.(z.t).\rho' \end{array} \right\}$$

Definition 2. For $P \subseteq Proc$, an abstract system $A \mathrel{\widehat{=}} (AI, (AOP_p)_{p:P}, AF)_{N,Z}$ is *data refined* by a concrete system $C \mathrel{\widehat{=}} (CI, (COP_p)_{p:P}, CF)_{N,Z}$, denoted $A \sqsubseteq C$ iff $obs_N.C \subseteq obs_N.A$.

It is trivial to prove that $\sqsubseteq$ is a preorder (i.e., a reflexive, transitive relation).

Verification of Definition 2 directly is infeasible. In state-based formalisms, data refinement is proved using *simulation*, which allows executions of the concrete system to be matched to executions of the abstract [30] (see Figure 3). Previous work [16, 18] defines operation refinement over a single state space. This cannot be used for example to prove refinement between the programs in Figures 1 and 2. In this paper, we develop simulation-based techniques for our interval-based framework in Section 4.3. The theory is based on interval relations (Section 4.2), which enable one to relate streams over two potentially different state spaces.

4.2 Interval relations

Interval predicates enable one to reason about properties that take time, however, only define properties over a single state space. Proving data refinement via simulation requires one to relate behaviours over a concrete state space to behaviours over an abstract space. Hence, we combine the ideas behind state relations and interval predicates and obtain *interval relations*, which are relations over an interval and two streams over potentially different state spaces. The concept of interval relations is novel to this paper.

An *interval relation* over Y and Z relates streams of Y and Z over intervals and is a mapping of type $IntvRel_{Y,Z} \mathrel{\widehat{=}} Intv \to Stream_Y \to Stream_Z \to \mathbb{B}$. Figure 7 depicts a visualisation of an interval relation over $Y, Z \subseteq Var$ where $z \in Stream_Z$ and $y \in Stream_Y$. Like interval predicates, we assume pointwise lifting of operators over state and interval relations in the normal manner. We extend interval predicate operators to interval relations, for example:

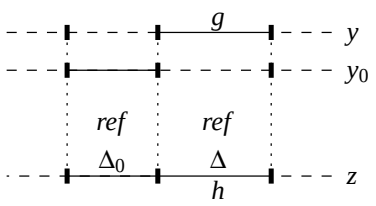


Figure 8: Visualisation of $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$

$$(R_1 ; R_2).\Delta.y.z \;\widehat{=}\; \big(\exists \Delta_1, \Delta_2 : Intv \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge R_1.\Delta_1.y.z \wedge R_2.\Delta_2.y.z\big) \vee (\mathsf{infinite}.\Delta \wedge R_1.\Delta.y.z)$$

A *state relation* over $Y, Z \subseteq Var$ is defined by its characteristic function $StateRel_{Y,Z} \mathrel{\widehat{=}} State_Y \to State_Z \to \mathbb{B}$. Operators on state predicates may be extended to state relations, e.g., for $r \in StateRel_{Y,Z}$ we define

$$\boxdot r.\Delta.y.z \;\widehat{=}\; \forall t : \Delta \bullet r.(y.t).(z.t)$$

If $R_1 \in IntvRel_{X,Y}$ and $R_2 \in IntvRel_{Y,Z}$ then for $\Delta \in Intv$, $x \in Stream_X$, $y \in Stream_Y$, we define the composition of R_1 and R_2 as

$$(R_1 \circ R_2).\Delta.x.z \;\widehat{=}\; \exists y : Stream_Y \bullet R_1.\Delta.x.y \wedge R_2.\Delta.y.z$$

4.3 Generalised forward simulation

In this section, we work towards an interval-based notion of forward simulation, which is then shown to be a sufficient condition for proving data refinement (Definition 2).

We define simulation between abstract and concrete systems with respect to an interval relation over the sets of representation variables of the two systems. This definition requires that we define equivalence between two streams over an interval. For streams y and z and interval Δ, we define a function

$$y \overset{\Delta}{=\!=} z \;\widehat{=}\; (\Delta \lhd y = \Delta \lhd z)$$

where '$\lhd$' denotes domain restriction. Thus $y \overset{\Delta}{=\!=} z$ holds iff the states of y and z corresponding to Δ match, i.e., $\forall t : \Delta \bullet y.t = z.t$. For $Y, Z \subseteq Var$, assuming that $g \in IntvPred_Y$ and $h \in IntvPred_Z$ model the abstract and concrete systems, respectively, and that $ref \in IntvRel_{Y,Z}$ denotes the refinement relation, we define a function $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$ (see Figure 8), which denotes that *h simulates g* with respect to *ref*.

$$ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right| \;\widehat{=}\; \begin{array}{l} \forall z : Stream_Z, \Delta, \Delta_0 : Intv, y_0 : Stream_Y \bullet \\ \quad (\Delta_0 \propto \Delta) \wedge ref.\Delta_0.y_0.z \wedge h.\Delta.z \Rightarrow \\ \quad \exists y : Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref.\Delta.y.z \wedge g.\Delta.y \end{array}$$

Thus, if $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$ holds, then for every concrete stream z, interval Δ and abstract state y, provided that

1. Δ_0 is an interval that immediately precedes Δ,
2. *ref* holds in the interval Δ_0 between y_0 and z, and
3. the concrete system (modelled by h) executes within Δ in stream z

then there exists an abstract stream y that matches y_0 over Δ_0 such that

1. the abstract system executes over Δ in y, and

2. *ref* holds between y and z over Δ.

A visualisation of $ref \bullet \left|\frac{Y \bullet g}{Z \bullet h}\right|$ is given in Figure 8 and is akin to matching a single non-stuttering concrete step to an abstract step in state-based forward simulation [30]. The following lemma establishes reflexivity and transitivity properties for $ref \bullet \left|\frac{Y \bullet g}{Z \bullet h}\right|$.

Lemma 3. *Provided that* $id.\sigma.\tau \mathrel{\widehat{=}} \sigma = \tau$.

$$\boxdot id \bullet \left|\frac{X \bullet g}{X \bullet g}\right| \qquad \text{(Reflexivity)}$$

$$ref_1 \bullet \left|\frac{X \bullet f}{Y \bullet g}\right| \wedge ref_2 \bullet \left|\frac{Y \bullet g}{Z \bullet h}\right| \;\Rightarrow\; (ref_1 \circ ref_2) \bullet \left|\frac{X \bullet f}{Z \bullet h}\right| \qquad \text{(Transitivity)}$$

Simulation is used to define an interval-based notion of *forward simulation* as follows.

Definition 4 (Forward simulation)**.** Suppose $P \subseteq Proc$, $A \mathrel{\widehat{=}} (AI, (AOp_p)_{p:P}, AF)_{N,Y}$ is an abstract system, $C \mathrel{\widehat{=}} (CI, (COp_p)_{p:P}, CF)_{N,Z}$ is a concrete system, and $ref \in IntvRel_{Y,Z}$. We say *ref* is a *forward simulation* from A to C iff $ref \bullet \left|\frac{Y \bullet \bigwedge_{p:P} AOp_p}{Z \bullet \bigwedge_{p:P} COp_p}\right|$ and both of the following hold:

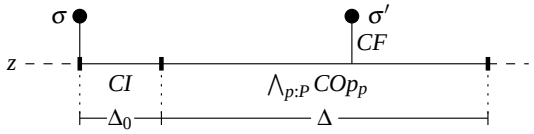
$$\forall z{:}Stream_Z, \Delta{:}Intv, \sigma \in State_N \bullet CI.\sigma.\Delta.z \;\Rightarrow\; \exists y{:}Stream_Y \bullet AI.\sigma.\Delta.y \wedge ref.\Delta.y.z \quad (10)$$

$$\forall z{:}Stream_Z, y{:}Stream_Y, \Delta{:}Intv, \sigma{:}State_N \bullet \forall t{:}\Delta \bullet$$
$$ref.\Delta.y.z \wedge CF.(z.t).\sigma \;\Rightarrow\; AF.(y.t).\sigma \quad (11)$$

The following theorem establishes soundness of our forward simulation rule with respect to interval-based data refinement.

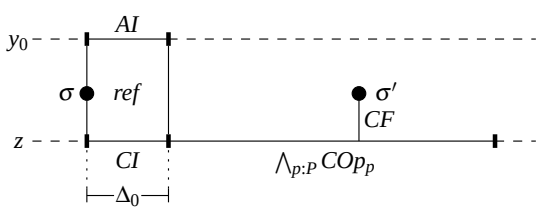
Theorem 5 (Soundness)**.** *If* $P \subseteq Proc$, $A \mathrel{\widehat{=}} (AI, (AOp_p)_{p:P}, AF)_{N,Y}$, *and* $C \mathrel{\widehat{=}} (CI, (COp_p)_{p:P}, CF)_{N,Z}$, *then* $A \sqsubseteq C$ *provided there exists a* $ref \in IntvRel_{Y,Z}$ *such that ref is a forward simulation from A to C.*

Proof. Suppose $\sigma, \sigma' \in State_N$, $z \in Stream_Z$ and C has an execution depicted below, where CI executes in interval Δ_0 and $\bigwedge_{p:P} COp_p$ executes in Δ. Note that $\bigwedge_{p:P} COp_p$ may or may not terminate, and hence, Δ may be infinite. To prove $A \sqsubseteq C$, it suffices to prove that there exists a matching execution of A starting in σ and ending in σ'.

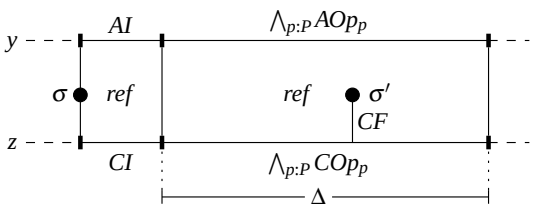


By (10), there exists a $y_0 \in Stream_Y$ such that $AI.\sigma.\Delta_0.y_0$ and $ref.\Delta_0.y_0.z$ hold recalling that Δ_0 is the initial interval of execution. This is depicted in (A) below. Now, because the simulation $ref \bullet \left|\frac{Y \bullet \bigwedge_{p:P} AOp_p}{Z \bullet \bigwedge_{p:P} COp_p}\right|$ holds, there exists a y that matches y_0 over Δ_0 such that both $(\bigwedge_{p:P} AOp_p).\Delta.y$ and $ref.\Delta.y.z$ hold, as depicted in (B) below.

(A)

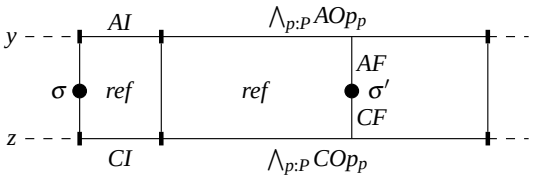


(B)



Then, due to the finalisation assumption (11), there exists a finalisation of A that results in σ' as shown in (D) below.

(D)



□

4.4 Decomposing simulations

A benefit of state-based forward simulation [30] is the ability to decompose proofs and focus on individual steps of the concrete system. Proof obligation $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$ in the interval-based forward simulation definition (Definition 4) takes the entire interval of execution of the concrete and abstract systems into account. Hence, we develop a number of methods for simplifying proofs of $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$. Decomposing $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$ directly is difficult due to the existential quantification in the consequent. However, a formula of the form $p \Rightarrow (\exists x \bullet q \wedge r)$ holds if both $p \Rightarrow \exists x \bullet q$ and $\forall x \bullet p \wedge q \Rightarrow r$ hold. Hence, we obtain the following lemma.

Lemma 6. *For any $Y, Z \subseteq Var$ and $ref \in IntvRel_{Y,Z}$, $ref \bullet \left|\!\!\frac{Y \bullet g}{Z \bullet h}\!\!\right|$ holds if both of the following hold:*

$$\begin{array}{rcll} \forall z{:}\,Stream_Z, \Delta, \Delta_0{:}\,Intv, y_0{:}\,Stream_Y \bullet & & & \\ \quad \Delta_0 \propto \Delta \wedge ref.\Delta_0.y_0.z \wedge h.\Delta.z & \Rightarrow & \exists y{:}\,Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref.\Delta.y.z & (12) \\ \forall z{:}\,Stream_Z, \Delta{:}\,Intv, y{:}\,Stream_Y \bullet & & & \\ \quad ref.\Delta.y.z \wedge h.\Delta.z & \Rightarrow & g.\Delta.y & (13) \end{array}$$

By (12), if the refinement predicate *ref* holds for an abstract stream y_0 in an immediately preceding interval Δ_0 and the concrete system executes in the current interval Δ, then there exists an abstract stream that matches y_0 over Δ_0 and *ref* holds for y over Δ. By (13) for any abstract stream y, concrete stream z and interval Δ, if the concrete system executes in Δ and forward simulation holds between y and z for Δ, then the behaviour of the abstract system holds for Δ in y.

To simplify representation of intervals of the form in (12), we introduce the following notation.

$$h \Vdash_{Y,Z} ref \quad \widehat{=} \quad (12)$$

The following lemma allows one to decompose proofs of the form given in $h \Vdash_{Y,Z} ref$.

Lemma 7. *If $Y, Z \subseteq Var$, $g, g_1, g_2 \in IntvPred_Z$ and $ref \in IntvRel_{Y,Z}$, then each of the following holds.*

$$\begin{array}{rcll} g_1 \Vdash_{Y,Z} ref \wedge g_2 \Vdash_{Y,Z} ref & \Rightarrow & (g_1 \,;\, g_2) \Vdash_{Y,Z} ref & \textit{provided ref joins} \quad \text{(Sequential composition)} \\ g \Vdash_{Y,Z} ref & \Rightarrow & g^{\omega} \Vdash_{Y,Z} ref & \textit{provided ref joins} \quad \text{(Iteration)} \\ (g_2 \Vdash_{Y,Z} ref) \wedge (g_1 \Rightarrow g_2) & \Rightarrow & g_1 \Vdash_{Y,Z} ref & \text{(Weaken)} \\ (g \Vdash_{Y,Z} ref_1) \vee (g \Vdash_{Y,Z} ref_2) & \Rightarrow & g \Vdash_{Y,Z} (ref_1 \vee ref_2) & \text{(Disjunction)} \end{array}$$

Note that *ref* can neither be weakened nor strengthened in the trivial manner because it appears in both the antecendent and consequent of the implication. If a refinement relation operates on two disjoint portions of the stream, it is possible to split the refinement as follows:

Lemma 8 (Disjointness)**.** *Suppose $p \in Proc$, $W,X,Y,Z \subseteq Var$ such that $Y \cap Z = \varnothing$, $W \cup X = Y$ and $W \cap X = \varnothing$. If $g_1, g_2 \in IntvPred_Z$, $ref_W \in IntvRel_{W,Z}$, $ref_X \in IntvRel_{X,Z}$, and $\star \in \{\wedge, \vee\}$, then*

$$(g_1 \Vdash_{W,Z} ref_W) \wedge (g_2 \Vdash_{X,Z} ref_X) \;\Rightarrow\; (g_1 \wedge g_2) \Vdash_{Y,Z} (ref_W \star ref_X) \qquad \text{(Disjointness)}$$

Disjointness allows one to prove mixed refinement, where the system states are split into disjoint subsets and different refinement relations are used to verify refinement between these substates.

Proof obligation (13) may also be simplified. In particular, for interval predicate g, interval Δ and streams y and z, we define $(g \restriction 1).\Delta.y.z \mathrel{\widehat{=}} g.\Delta.y$ and $(g \restriction 2).\Delta.y.z \mathrel{\widehat{=}} g.\Delta.z$, which allows one to shorten (13) to

$$ref \wedge (h \restriction 2) \;\Rightarrow\; (g \restriction 1) \tag{14}$$

Hence, proofs of refinement are reduced to proofs of implication between the concrete and abstract state spaces. There are numerous rules for decomposing proofs of the form in (14) that exploit rely/guarantee-style reasoning [20, 16].

5 Fine-grained atomicity

Interval-based reasoning provides the opportunity to incorporate methods for non-deterministically evaluating expressions [8, 25], which captures the possible low-level interleavings (e.g., Figure 4) at a higher-level of abstraction. Methods for non-deterministically evaluating expressions are given in Section 5.1, and also appear in [8, 25, 19, 18, 20]. Verification of data refinement of our running example that combines non-deterministic evaluation from Section 5.1 and the data refinement rules from Section 4 is given in Section 5.2.

5.1 Non-deterministically evaluating expressions

Most hardware can only guarantee that at most one global variable can be read in a single atomic step. Thus, in the presence of possibly interfering processes and fine-grained atomicity, a model that assumes expressions containing multiple shared variables can be evaluated in a single state may not be implementable without the introduction of contention inducing locks [1, 3, 27]. As we have done in Figure 4, one may split expression evaluation into a number of atomic steps to make the underlying atomicity explicit. However, this approach is undesirable as it causes the complexity of expression evaluation to increase exponentially with the number of variables in an expression — evaluation of an expression with n (global) variables would require one to check $n!$ permutations of the read order.

Interval-based reasoning enables one to incorporate methods for non-deterministically evaluating state predicates over an evaluation interval [25], which allow the possible permutations in the read order of variables to be considered at a high level of abstraction. For this paper, we use *apparent states evaluators*, which allow one to evaluate an expression e with respect to the set of states that are apparent to a process. Each variable of e is assumed to be read at most once, but at potentially different instants, and hence, instead of evaluating e in a single atomic step, apparent states evaluations assume expression evaluation takes time and considers the set of states that occur over the interval of evaluation. An apparent

state is generated by picking a value for each variable from the set of actual values of the variable over the interval of evaluation. For $\Delta \in \mathit{Intv}$ and $s \in \mathit{Stream}_V$, we define:

$$\mathit{apparent}.\Delta.s \;\widehat{=}\; \{\sigma{:}\,\mathit{State}_V \mid \forall v{:}\,V \bullet \exists t{:}\,\Delta \bullet \sigma.v = s.t.v\}$$

Example 9. Consider the statements $u := 1\,;\; v := 1$ which we assume are executed over an interval Δ from an initial state that satisfies $u,v = 0,0$. The set of states that actually occur over this interval is hence

$$\{\{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}\}$$

Evaluation of $u < v$ in the set of actual states above always results in *false*. Assuming no other (parallel) modifications to u and v, for some stream s over $\{u,v\}$, the set of apparent states corresponding to Δ is:

$$\mathit{apparent}.\Delta.s \;=\; \big\{\{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}, \{u \mapsto 0, v \mapsto 1\}, \{u \mapsto 1, v \mapsto 0\}\big\}$$

where the additional apparent state $\{u \mapsto 0, v \mapsto 1\}$ may be obtained by reading u with value 0 (in the initial state) and v with value 1 (after both modifications). Unlike the actual states evaluation, $u < v$ may result in *false* when evaluating in the apparent states. Note that $v = v$ still only has one possible value, *true*, i.e., apparent states evaluation assumes that the same value of v is used for both occurrences of v.

Two useful operators for a sets of apparent states evaluation allow one to formalise that *c definitely* holds (denoted $\circledast c$) and *c possibly* holds (denoted $\circledcirc c$), which are defined as follows.

$$(\circledast c).\Delta.s \;\widehat{=}\; \forall \sigma{:}\,\mathit{apparent}.\Delta.s \bullet c.\sigma \qquad\qquad (\circledcirc c).\Delta.s \;\widehat{=}\; \exists \sigma{:}\,\mathit{apparent}.\Delta.s \bullet c.\sigma$$

The following lemma states a relationship between *definitely* and *always* properties, as well as between *possibly* and *sometime* properties [25]. Note that both $\circledast c \Rightarrow \boxdot c$ and $\diamondsuit c \Rightarrow \circledcirc c$ hold, but the converse of both properties are not necessarily true.

Example 10. We now instantiate the guard evaluations of the form $[c]$ within (6) and (7). In particular, a guard c holds if it is possible to evaluate the variables of c (at potentially different instants) so that c evaluates to *true*. Therefore, the semantics of the evaluation of a guard c is formalised by $\circledcirc c$ and we obtain the following interval predicates for (6) and (7).

$$(\circledcirc \mathit{grd}\,;\; \underline{\boxdot(m = 1)} \vee \circledcirc \neg \mathit{grd}\,;\; \underline{\boxdot(m = 2)}) \;\wedge\; (\circledcirc b\,;\; \underline{\boxdot \mathit{grd}} \vee \circledcirc \neg b) \tag{15}$$

$$\begin{pmatrix} \circledcirc(u < v)\,;\; \underline{\boxdot(m = 1)} \vee \\ \circledcirc(u \geq v)\,;\; \underline{\boxdot(m = 2)} \end{pmatrix} \;\wedge\; \begin{pmatrix} \circledcirc(0 < u)\,;\; \underline{\boxdot(v = \infty)} \vee \\ \circledcirc(0 \geq u)\,;\; \underline{\boxdot(v = -\infty)} \end{pmatrix} \tag{16}$$

Note that interval predicate $\circledcirc(u < v)$ is equivalent to

$$\exists k_u, k_v \bullet ((\diamondsuit(k_u = u)\,;\; \diamondsuit(k_v = v)) \vee (\diamondsuit(k_v = v)\,;\; \diamondsuit(k_u = u)))\,;\; (k_u < k_v)$$

Hence, the formalisation in (16) accurately captures the fine-grained behaviour of Figure 2 without having to explicitly decompose the guard evaluation at cp_1 into individual reads as done in Figure 4.

The theory in [25] allows one to relate different forms of non-deterministic evaluation. For example, both $\circledast c \Rightarrow \boxdot c$ and $\diamondsuit c \Rightarrow \circledcirc c$ hold. To strengthen the implication to an equivalence, one must introduce additional assumptions about the stability of the variables of c. Because adjoining intervals are disjoint, the definition of stability must refer to the value of c at the end of an immediately preceding interval [18, 16, 20]. For a state predicate c, interval Δ and stream s, we define

$$\mathit{prev}.c.\Delta.s \;\widehat{=}\; \exists \Delta'{:}\,\mathit{Intv} \bullet \Delta' \propto \Delta \wedge \underline{\boxdot c}.\Delta'.s$$

Variable v is stable over a Δ in s (denoted $\mathsf{stable}.v.\Delta.s$) iff the value of v does not change from its value over some interval that immediately precedes Δ. A set of variables V is *stable* in Δ (denoted $\mathsf{stable}.V.\Delta$) iff each variable in V is stable in Δ. Thus, we define:

$$\mathsf{stable}.v.\Delta.s \;\widehat{=}\; \exists k{:}\,\mathit{Val} \bullet (\mathit{prev}.(v = k) \wedge \boxdot(v = k)).\Delta.s \qquad\qquad \mathsf{stable}.V.\Delta \;\widehat{=}\; \forall v{:}\,V \bullet \mathsf{stable}.v.\Delta$$

Note that every variable is stable in an empty interval and the empty set of variables is stable in any interval, i.e., both $\mathsf{stable}.V.\varnothing$ and $\mathsf{stable}.\varnothing.\Delta$ hold trivially.

We let $vars.c$ denote the free variables of state predicate c. The following lemma states that if all but one variable of c is stable over an interval Δ, then c definitely holds in Δ iff c always holds in Δ, and that c possibly holds in Δ iff c holds sometime in Δ [25].

Lemma 11. *For a state predicate c and variable v,* $\mathsf{stable}.(vars.c\backslash\{v\}) \Rightarrow (\circledast c = \boxdot c) \wedge (\circledcirc c = \diamondsuit c)$.

Example 12. For our running example, by Lemma 11, it is possible to simplify (15) and (16) and replace each occurrence of '$\circledcirc$' by '$\diamondsuit$' as follows:

$$(\diamondsuit grd\,;\ \underline{\boxdot(m=1)} \vee \diamondsuit\neg grd\,;\ \underline{\boxdot(m=2)}) \ \wedge\ (\diamondsuit b\,;\ \underline{\boxdot grd} \vee \diamondsuit\neg b) \qquad \text{(Abs-IP)}$$

$$\begin{pmatrix}\diamondsuit(u<v)\,;\ \underline{\boxdot(m=1)} \vee\\ \diamondsuit(u\geq v)\,;\ \underline{\boxdot(m=2)}\end{pmatrix} \ \wedge\ \begin{pmatrix}\diamondsuit(0<u)\,;\ \underline{\boxdot(v=\infty)} \vee\\ \diamondsuit(0\geq u)\,;\ \underline{\boxdot(v=-\infty)}\end{pmatrix} \qquad \text{(Conc-IP)}$$

5.2 Data refinement example

We assume the representation variables of the abstract and concrete programs are given by $Y \subseteq Var$ and $Z \subseteq Var$, respectively and prove forward simulation using $\underline{\boxdot uv}$ (recalling that relation uv is defined in Section 2), which requires that we prove

$$\underline{\boxdot uv} \bullet \left|\begin{array}{c} Y \bullet \text{(Abs-IP)} \\ \hline\hline Z \bullet \text{(Conc-IP)} \end{array}\right| \tag{17}$$

and both of the following:

$$\begin{array}{rcl} \forall \Delta{:}\,Intv, z{:}\,Stream_Z, \sigma{:}\,State_N \bullet & & \\ \quad CInit.\sigma.\Delta.z & \Rightarrow & \exists y{:}\,Stream_Y \bullet AInit.\sigma.\Delta.y \wedge \underline{\boxdot uv}.\Delta.y.z \end{array} \tag{18}$$

$$\begin{array}{rcl} \forall z{:}\,Stream_Z, y{:}\,Stream_Y, \Delta{:}\,Intv, \sigma{:}\,State_N \bullet \forall t{:}\,\Delta \bullet & & \\ \quad \underline{\boxdot uv}.\Delta.y.z \wedge CFin.(z.t).\sigma & \Rightarrow & AFin.(y.t).\sigma \end{array} \tag{19}$$

The proofs of (18) and (19) are trivial. To prove (17), we use Lemma 6, which requires that we show that both of the following hold. Recall that uv is the state relation defined in Section 3.

$$(\text{Conc-IP}) \Vdash_{Y,Z} \underline{\boxdot uv} \tag{20}$$

$$\underline{\boxdot uv} \wedge (\text{Conc-IP}) \restriction 2 \Rightarrow (\text{Abs-IP}) \restriction 1 \tag{21}$$

The proof of (20) is trivial. Expanding the definitions of (Abs-IP) and (Conc-IP), then applying some straightforward propositional logic, (21), holds if both of the following hold.

$$\underline{\boxdot uv} \wedge \begin{pmatrix}\diamondsuit(0<u)\,;\ \underline{\boxdot(v=\infty)} \vee\\ \diamondsuit(0\geq u)\,;\ \underline{\boxdot(v=-\infty)}\end{pmatrix} \restriction 2 \ \Rightarrow\ (\diamondsuit b\,;\ \underline{\boxdot grd} \vee \diamondsuit\neg b) \restriction 1 \tag{22}$$

$$\underline{\boxdot uv} \wedge \begin{pmatrix}\diamondsuit(u<v)\,;\ \underline{\boxdot(m=1)} \vee\\ \diamondsuit(u\geq v)\,;\ \underline{\boxdot(m=2)}\end{pmatrix} \restriction 2 \ \Rightarrow\ \begin{pmatrix}\diamondsuit grd\,;\ \underline{\boxdot(m=1)} \vee\\ \diamondsuit\neg grd\,;\ \underline{\boxdot(m=2)}\end{pmatrix} \restriction 1 \tag{23}$$

Condition (22) is proved in a straightforward manner as follows and uses the fact that $\boxdot(u<\infty)$ holds throughout the execution of Figure 2.

$$
\begin{array}{ll}
 & \underline{\boxdot uv} \wedge (\diamondsuit(0 < u)\,;\, \underline{\boxdot(v = \infty)} \vee \diamondsuit(0 \geq u)\,;\, \underline{\boxdot(v = -\infty)}) \restriction 2 \\
\Rightarrow & \quad \text{distribute projection, logic and } \boxdot(u < \infty) \\
 & \boxdot uv \wedge ((\diamondsuit(0 < u) \restriction 2\,;\, \underline{\boxdot(u < v)} \restriction 2) \vee (\diamondsuit(0 \geq u) \restriction 2\,;\, \underline{\boxdot(u \geq v)} \restriction 2)) \\
\Rightarrow & \quad \text{distribute } \wedge, \boxdot uv \text{ splits} \\
 & (\boxdot uv \wedge \diamondsuit(0 < u) \restriction 2))\,;\, (\boxdot uv \wedge \underline{\boxdot(u < v)} \restriction 2) \vee \\
 & (\boxdot uv \wedge \diamondsuit(0 \geq u) \restriction 2)\,;\, (\boxdot uv \wedge \underline{\boxdot(u \geq v)} \restriction 2) \\
\Rightarrow & \quad \text{use } \boxdot uv \\
 & (\diamondsuit b \restriction 1\,;\, \underline{\boxdot grd} \restriction 1) \vee (\diamondsuit \neg b) \restriction 1 \\
= & \quad \text{distribute projection} \\
 & ((\diamondsuit b\,;\, \underline{\boxdot grd}) \vee \diamondsuit \neg b) \restriction 1
\end{array}
$$

The proof of (23) has a similar structure, and hence, its details are elided.

The example verification demonstrates many of the benefits of using interval-based reasoning to prove data refinement between concurrent systems. The proofs themselves are succinct (and consequently more understandable) because the reasoning is performed at a high level of abstraction. Expression evaluation is assumed to take time and evaluation operators such as '$\diamondsuit$' and '$\diamondsuit\!\!\!\!\diamondsuit$' are used to capture the inherent non-determinism that results from concurrent executions during the interval of evaluation. Furthermore, the translation of the program in Figure 2 to the lower-level program Figure 4 that makes the non-determinism for evaluating reads explicit is not necessary. Instead, one is able to provide a semantics for the program in Figure 2 directly. Finally, unlike a state-based forward simulation proof, which requires that a verifier explicitly decides which of the concrete steps are non-stuttering, then find a corresponding abstract step for each non-stuttering step, interval-based reasoning allows one to remove this analysis step altogether.

6 Conclusions

Interval-based frameworks are effective for reasoning about fine-grained atomicity and true concurrency in the presence of both discrete and continuous properties. The main contribution of this paper is the development of generalised methods for proving data refinement using interval-based reasoning. A simulation rule for proving data refinement is developed and soundness of the rule with respect to the data refinement definition is proved. Our simulation rule allows the use of refinement relations between streams over two state spaces within an interval, generalising traditional refinement relations, which only relate two states. Using interval-based reasoning enables one to incorporate methods for non-deterministically evaluating expressions, which in combination with our simulation rules are used to verify data refinement of a simple concurrent program.

Over the years, numerous theories for data refinement have been developed. As far as we are aware, two of these are based on interval-based principles similar to ours. A framework that combines interval temporal logic and refinement has been defined by Bäumler et al [5], but their execution model explicitly interleaves a component and its environment. As a result, our high-level expression evaluation operators cannot be easily incorporated into their framework. Furthermore, refinement is defined in terms of relations between the abstract and concrete states. Broy presents refinement between streams of different types of timed systems (e.g., discrete vs. continuous systems) [7]; however, these methods do not consider interval-based reasoning. An interesting direction of future work would be to consider a model that combines our methods with theories for refinement between different abstractions of time.

Acknowledgements This work is sponsored by EPSRC Grant EP/J003727/1. We thank our anonymous

reviewers for their numerous insightful comments. In particular, one reviewer who pointed out a critical flaw in one of our lemmas.

References

[1] M. Abadi & L. Lamport (1995): *Conjoining Specifications*. *ACM Trans. Program. Lang. Syst.* 17(3), pp. 507–534. Available at http://doi.acm.org/10.1145/203095.201069.

[2] R.-J. Back & J. von Wright (1994): *Trace Refinement of Action Systems*. In B. Jonsson & J. Parrow, editors: *CONCUR, LNCS* 836, Springer, pp. 367–384. Available at http://dx.doi.org/10.1007/BFb0015020.

[3] R.-J. Back & J. von Wright (1999): *Reasoning algebraically about loops*. *Acta Informatica* 36(4), pp. 295–334, doi:10.1007/s002360050163.

[4] P. Baldan & S. Crafa (2010): *A Logic for True Concurrency*. In P. Gastin & F. Laroussinie, editors: *CONCUR, LNCS* 6269, Springer, pp. 147–161. Available at http://dx.doi.org/10.1007/978-3-642-15375-4_11.

[5] S. Bäumler, G. Schellhorn, B. Tofan & W. Reif (2011): *Proving linearizability with temporal logic*. *Formal Asp. Comput.* 23(1), pp. 91–112. Available at http://dx.doi.org/10.1007/s00165-009-0130-y.

[6] J. Boyland (2003): *Checking Interference with Fractional Permissions*. In R. Cousot, editor: *SAS, LNCS* 2694, Springer, pp. 55–72. Available at http://dx.doi.org/10.1007/3-540-44898-5_4.

[7] M. Broy (2001): *Refinement of time*. *Theor. Comput. Sci.* 253(1), pp. 3–26. Available at http://dx.doi.org/10.1016/S0304-3975(00)00087-6.

[8] A. Burns & I. J. Hayes (2010): *A timeband framework for modelling real-time systems*. *Real-Time Systems* 45(1-2), pp. 106–142. Available at http://dx.doi.org/10.1007/s11241-010-9094-5.

[9] J. Derrick & E. A. Boiten (1999): *Non-atomic Refinement in Z*. In J. M. Wing, J. Woodcock & J. Davies, editors: *World Congress on Formal Methods, LNCS* 1709, Springer, pp. 1477–1496. Available at http://dx.doi.org/10.1007/3-540-48118-4_28.

[10] J. Derrick & E. A. Boiten (2003): *Relational Concurrent Refinement*. *Formal Asp. Comput.* 15(2-3), pp. 182–214. Available at http://dx.doi.org/10.1007/s00165-003-0007-4.

[11] J. Derrick & E. A. Boiten (2007): *Relational Concurrent Refinement with Internal Operations*. *Electr. Notes Theor. Comput. Sci.* 187, pp. 35–53. Available at http://dx.doi.org/10.1016/j.entcs.2006.08.043.

[12] J. Derrick & E. A. Boiten (2009): *Relational Concurrent Refinement: Automata*. *Electr. Notes Theor. Comput. Sci.* 259, pp. 21–34. Available at http://dx.doi.org/10.1016/j.entcs.2009.12.015.

[13] J. Derrick, S. Gnesi, D. Latella & H. Treharne, editors (2012): *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. *LNCS* 7321, Springer. Available at http://dx.doi.org/10.1007/978-3-642-30729-4.

[14] B. Dongol (2009): *Progress-based verification and derivation of concurrent programs*. Ph.D. thesis, The University of Queensland.

[15] B. Dongol & J. Derrick (2012): *Proving linearisability via coarse-grained abstraction*. *CoRR* abs/1212.5116. Available at http://arxiv.org/abs/1212.5116.

[16] B. Dongol, J. Derrick & I. J. Hayes (2012): *Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic*. *ECEASST* 53. Available at http://journal.ub.tu-berlin.de/eceasst/article/view/792.

[17] B. Dongol & I. J. Hayes (2010): *Compositional Action System Derivation Using Enforced Properties*. In C. Bolduc, J. Desharnais & B. Ktari, editors: *MPC, LNCS* 6120, Springer, pp. 119–139. Available at http://dx.doi.org/10.1007/978-3-642-13321-3_9.

[18] B. Dongol & I. J. Hayes (2012): *Deriving Real-Time Action Systems Controllers from Multiscale System Specifications*. In J. Gibbons & P. Nogueira, editors: *MPC, LNCS* 7342, Springer, pp. 102–131. Available at http://dx.doi.org/10.1007/978-3-642-31113-0_7.

[19] B. Dongol & I. J. Hayes (2012): *Deriving real-time action systems in a sampling logic*. *Science of Computer Programming* (0), pp. –, doi:10.1016/j.scico.2012.07.008. Available at http://www.sciencedirect.com/science/article/pii/S0167642312001360.

[20] B. Dongol & I. J. Hayes (2012): *Rely/Guarantee Reasoning for Teleo-reactive Programs over Multiple Time Bands*. In Derrick et al. [13], pp. 39–53. Available at http://dx.doi.org/10.1007/978-3-642-30729-4_4.

[21] B. Dongol, I. J. Hayes, L. Meinicke & K. Solin (2012): *Towards an Algebra for Real-Time Programs*. In W. Kahl & T. G. Griffin, editors: *RAMICS, LNCS* 7560, Springer, pp. 50–65. Available at http://dx.doi.org/10.1007/978-3-642-33314-9_4.

[22] W. H. J. Feijen & A. J. M. van Gasteren (1999): *On a Method of Multiprogramming*. Springer Verlag.

[23] C. J. Fidge (1993): *Real-Time Refinement*. In J. Woodcock & P. G. Larsen, editors: *FME, Lecture Notes in Computer Science* 670, Springer, pp. 314–331. Available at http://dx.doi.org/10.1007/BFb0024654.

[24] C. J. Fidge, M. Utting, P. Kearney & I. J. Hayes (1996): *Integrating Real-Time Scheduling Theory and Program Refinement*. In M.-C. Gaudel & J. Woodcock, editors: *FME, LNCS* 1051, Springer, pp. 327–346. Available at http://dx.doi.org/10.1007/3-540-60973-3_95.

[25] I. J. Hayes, A. Burns, B. Dongol & C. B. Jones (2013): *Comparing Degrees of Non-Determinism in Expression Evaluation*. *The Computer Journal*, doi:10.1093/comjnl/bxt005. Available at http://comjnl.oxfordjournals.org/content/early/2013/02/05/comjnl.bxt005.abstract.

[26] E. C. R. Hehner (1990): *A Practical Theory of Programming*. *Sci. Comput. Program.* 14(2-3), pp. 133–158. Available at http://dx.doi.org/10.1016/0167-6423(90)90018-9.

[27] C. B. Jones & K. G. Pierce (2008): *Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification*. In E., M. J. Butler, J. P. Bowen & P. Boca, editors: *ABZ, Lecture Notes in Computer Science* 5238, Springer, pp. 360–377. Available at http://dx.doi.org/10.1007/978-3-540-87603-8_47.

[28] M. E. Majster-Cederbaum & J. Wu (2001): *Action Refinement for True Concurrent Real Time*. In: *ICECCS*, IEEE Computer Society, pp. 58–68. Available at http://doi.ieeecomputersociety.org/10.1109/ICECCS.2001.930164.

[29] Ben C. Moszkowski (2000): *A Complete Axiomatization of Interval Temporal Logic with Infinite Time*. In: *LICS*, IEEE Computer Society, pp. 241–252. Available at http://doi.ieeecomputersociety.org/10.1109/LICS.2000.855773.

[30] W. P. de Roever & K. Engelhardt (1996): *Data Refinement: Model-oriented proof methods and their comparison*. *Cambridge Tracts in Theoretical Computer Science* 47, Cambridge University Press.

[31] D. Scholefield, H. S. M. Zedan & J. He (1993): *Real-Time Refinement: Semantics and Application*. In A. M. Borzyszkowski & S. Sokolowski, editors: *MFCS, LNCS* 711, Springer, pp. 693–702. Available at http://dx.doi.org/10.1007/3-540-57182-5_60.

[32] C. Zhou & M. R. Hansen (2004): *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science, Springer.

A Proofs of lemmas

Lemma (3) Provided that $id.\sigma.\tau \mathrel{\widehat{=}} \sigma = \tau$.

$$\boxdot id \bullet \left|\!\!\begin{array}{c} X \bullet g \\ \hline X \bullet g \end{array}\!\!\right| \qquad \text{(Reflexivity)}$$

$$ref_1 \bullet \left|\!\!\begin{array}{c} X \bullet f \\ \hline Y \bullet g \end{array}\!\!\right| \wedge ref_2 \bullet \left|\!\!\begin{array}{c} Y \bullet g \\ \hline Z \bullet h \end{array}\!\!\right| \;\Rightarrow\; (ref_1 \circ ref_2) \bullet \left|\!\!\begin{array}{c} X \bullet f \\ \hline Z \bullet h \end{array}\!\!\right| \qquad \text{(Transitivity)}$$

Proof. The proof of (Reflexivity) is trivial. We prove (Transitivity) as follows, where we assume that $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $x_0 \in \mathit{Stream}_X$ and $z \in \mathit{Stream}_Z$ are arbitrarily chosen. We have:

$$
\begin{array}{ll}
 & (\mathit{ref}_1 \circ \mathit{ref}_2).\Delta_0.x_0.z \wedge h.\Delta.z \\
= & \quad \text{definition of } \circ \text{ and logic} \\
 & \exists y_0 : \mathit{Stream}_Y \bullet \mathit{ref}_1.\Delta_0.x_0.y_0 \wedge \mathit{ref}_2.\Delta_0.y_0.z \wedge h.\Delta.z
\end{array}
$$

Hence, for an arbitrarily chosen $y_0 \in \mathit{Stream}_Y$, we prove the following.

$$
\begin{array}{ll}
 & \mathit{ref}_1.\Delta_0.x_0.y_0 \wedge \mathit{ref}_2.\Delta_0.y_0.z \wedge h.\Delta.z \\
\Rightarrow & \quad \text{assumption } \mathit{ref}_2 \bullet \left|\!\!\begin{array}{c} Y \bullet g \\ \hline Z \bullet h \end{array}\!\!\right| \\
 & \mathit{ref}_1.\Delta_0.x_0.y_0 \wedge \exists y : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}_2.\Delta.y.z \wedge g.\Delta.y \\
= & \quad \text{logic assuming freeness of } y \\
 & \exists y : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}_2.\Delta.y.z \wedge \mathit{ref}_1.\Delta_0.x_0.y_0 \wedge g.\Delta.y \\
= & \quad \mathit{Stream}_Y \text{ contains all possible streams} \\
 & \exists y_1 : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge (y \overset{\Delta}{=\!=} y_1) \wedge \mathit{ref}_2.\Delta.y.z \wedge \mathit{ref}_1.\Delta_0.x_0.y_0 \wedge g.\Delta.y \\
= & \quad \text{use } y_0 \overset{\Delta_0}{=\!=} y_1 \text{ and } y \overset{\Delta}{=\!=} y_1 \\
 & \exists y_1 : \mathit{Stream}_Y \bullet \mathit{ref}_2.\Delta.y_1.z \wedge \mathit{ref}_1.\Delta_0.x_0.y_1 \wedge g.\Delta.y_1 \\
\Rightarrow & \quad \text{logic, assumption } \mathit{ref}_1 \bullet \left|\!\!\begin{array}{c} X \bullet f \\ \hline Y \bullet g \end{array}\!\!\right| \\
 & \exists y_1 : \mathit{Stream}_Y, x : \mathit{Stream}_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge \mathit{ref}_2.\Delta.y_1.z \wedge \mathit{ref}_1.\Delta.x.y_1 \wedge f.\Delta.x \\
\Rightarrow & \quad \text{definition of } \circ \\
 & \exists x : \mathit{Stream}_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge (\mathit{ref}_1 \circ \mathit{ref}_2).\Delta.x.z \wedge f.\Delta.x
\end{array}
$$

□

Lemma (7) Suppose $Y, Z \subseteq \mathit{Var}$ such that $Y \cap Z = \varnothing$, $g, g_1, g_2 \in \mathit{IntvPred}_Z$ and $\mathit{ref} \in \mathit{IntvRel}_{Y,Z}$. Then:

$$
\begin{array}{rcll}
g_1 \Vdash_{Y,Z} \mathit{ref} \wedge g_2 \Vdash_{Y,Z} \mathit{ref} & \Rightarrow & (g_1 \,;\, g_2) \Vdash_{Y,Z} \mathit{ref} & \text{provided } \mathit{ref} \text{ joins} \quad \text{(Sequential composition)} \\
g \Vdash_{Y,Z} \mathit{ref} & \Rightarrow & g^\omega \Vdash_{Y,Z} \mathit{ref} & \text{provided } \mathit{ref} \text{ joins} \quad \text{(Iteration)} \\
(g_2 \Vdash_{Y,Z} \mathit{ref}) \wedge (g_1 \Rightarrow g_2) & \Rightarrow & g_1 \Vdash_{Y,Z} \mathit{ref} & \text{(Weaken)} \\
(g \Vdash_{Y,Z} \mathit{ref}_1) \vee (g \Vdash_{Y,Z} \mathit{ref}_2) & \Rightarrow & g \Vdash_{Y,Z} (\mathit{ref}_1 \vee \mathit{ref}_2) & \text{(Disjunction)}
\end{array}
$$

Proof of (Sequential composition). For an arbitrarily chosen $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $y_0 \in \mathit{State}_Y$ and $z \in \mathit{Stream}_Z$, we have the following calculation.

$$
\begin{array}{ll}
 & \mathit{ref}.\Delta_0.y_0.z \wedge (g_1 \,;\, g_2).\Delta.z \\
= & \quad \text{definition of `;', logic} \\
 & \exists \Delta_1, \Delta_2 : \mathit{Intv} \bullet (\Delta_1 \cup \Delta_2 = \Delta) \wedge (\Delta_1 \propto \Delta_2) \wedge \mathit{ref}.\Delta_0.y_0.z \wedge g_1.\Delta_1.z \wedge g_2.\Delta_2.z \\
\Rightarrow & \quad \Delta_0 \propto \Delta \text{ and } \Delta_1 \in \mathit{prefix}.\Delta, \text{ therefore } \Delta_0 \propto \Delta_1 \\
 & \quad \text{assumption } g_1 \Vdash_{Y,Z} \mathit{ref} \\
 & \exists \Delta_1, \Delta_2 : \mathit{Intv} \bullet (\Delta_1 \cup \Delta_2 = \Delta) \wedge (\Delta_1 \propto \Delta_2) \wedge (\exists y_1 : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z) \wedge g_2.\Delta_2.z \\
= & \quad \text{logic} \\
 & \exists \Delta_1, \Delta_2 : \mathit{Intv}, y_1 : \mathit{Stream}_Y \bullet (\Delta_1 \cup \Delta_2 = \Delta) \wedge (\Delta_1 \propto \Delta_2) \wedge (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z \wedge g_2.\Delta_2.z \\
= & \quad \Delta_1 \propto \Delta_2 \text{ and assumption } g_2 \Vdash_{Y,Z} \mathit{ref} \\
 & \exists \Delta_1, \Delta_2 : \mathit{Intv}, y_1, y_2 : \mathit{Stream}_Y \bullet (\Delta_1 \cup \Delta_2 = \Delta) \wedge (\Delta_1 \propto \Delta_2) \wedge \\
 & \qquad\qquad (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z \wedge (y_1 \overset{\Delta_1}{=\!=} y_2) \wedge \mathit{ref}.\Delta_2.y_2.z \\
\Rightarrow & \quad \text{pick } y_3 \text{ such that } y_1 \overset{\Delta_0 \cup \Delta_1}{=\!=} y_3 \text{ and } y_2 \overset{\Delta_2}{=\!=} y_3
\end{array}
$$

$$
\begin{array}{ll}
 & \exists \Delta_1, \Delta_2 : \mathit{Intv}, y_3 : \mathit{Stream}_Y \bullet (\Delta_1 \cup \Delta_2 = \Delta) \wedge (\Delta_1 \propto \Delta_2) \wedge (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge \mathit{ref}.\Delta_1.y_3.z \wedge \mathit{ref}.\Delta_2.y_3.z \\
= & \quad \text{definition} \\
 & \exists y_3 : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge (\mathit{ref}\,;\,\mathit{ref}).\Delta.y_3.z \\
\Rightarrow & \quad \mathit{ref} \text{ joins} \\
 & \exists y_3 : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge \mathit{ref}.\Delta.y_3.z
\end{array}
$$

□

Proof of (Iteration). This followings by unfolding $^\omega$ and has a similar structure to (Sequential composition). □

Proof of (Weaken). For an arbitrarily chosen $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $y_0 \in \mathit{State}_Y$ and $z \in \mathit{Stream}_Z$, we have the following calculation.

$$
\begin{array}{ll}
 & \mathit{ref}.\Delta_0.y_0.z \wedge g_1.\Delta.z \\
\Rightarrow & \quad \text{assumption } g_1 \Rightarrow g_2 \\
 & \mathit{ref}.\Delta_0.y_0.z \wedge g_2.\Delta.z \\
\Rightarrow & \quad \text{assumption } g_2 \Vdash_{Y,Z} \mathit{ref} \\
 & \exists y : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}.\Delta.y.z
\end{array}
$$

□

Proof of (Disjunction).

$$
\begin{array}{ll}
 & (\mathit{ref}_1 \vee \mathit{ref}_2).\Delta_0.y_0.z \wedge g.\Delta.z \\
= & \quad \text{logic} \\
 & (\mathit{ref}_1.\Delta_0.y_0.z \wedge g.\Delta.z) \vee (\mathit{ref}_2.\Delta_0.y_0.z \wedge g.\Delta.z) \\
\Rightarrow & \quad \text{assumption } (g \Vdash_{Y,Z} \mathit{ref}_1) \vee (g \Vdash_{Y,Z} \mathit{ref}_2), \text{ logic} \\
 & \exists y_1, y_2 : \mathit{Stream}_Y \bullet ((y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}_1.\Delta.y_1.z) \vee ((y_0 \overset{\Delta_0}{=\!=} y_2) \wedge \mathit{ref}_2.\Delta.y_2.z) \\
\Rightarrow & \quad \text{logic} \\
 & \exists y : \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge (\mathit{ref}_1 \vee \mathit{ref}_2).\Delta.y.z
\end{array}
$$

□

For streams s_1 and s_2, we define $s_1 \Cup s_2 \mathrel{\widehat{=}} \lambda\, t : \Phi \bullet s_1.t \cup s_2.t$. If the state spaces corresponding to s_1 and s_2 are disjoint, then for each $t \in \Phi$, $(s_1 \Cup s_2).t$ is a state and hence $s_1 \Cup s_2$ is a stream.

Lemma (8)(Disjointness) Suppose $p \in \mathit{Proc}$, $W, X, Y, Z \subseteq \mathit{Var}$ such that $Y \cap Z = \varnothing$, $W \cup X = Y$ and $W \cap X = \varnothing$. Further suppose that $g_1, g_2 \in \mathit{IntvPred}_Z$, $\mathit{ref}_W \in \mathit{IntvRel}_{W,Z}$, $\mathit{ref}_X \in \mathit{IntvRel}_{X,Z}$, and $\star \in \{\wedge, \vee\}$. Then

$$
(g_1 \Vdash_{W,Z} \mathit{ref}_W) \wedge (g_2 \Vdash_{X,Z} \mathit{ref}_X) \;\Rightarrow\; (g_1 \wedge g_2) \Vdash_{Y,Z} (\mathit{ref}_W \star \mathit{ref}_X) \qquad \text{(Disjointness)}
$$

Proof. Because $W \cup X = Y$ and $W \cap X = Y$, for any $y_0 \in \mathit{Stream}_Y$, we have that $y_0 = w_0 \Cup x_0$ for some $w_0 \in \mathit{Stream}_W$, $x_0 \in \mathit{Stream}_X$. Then for any $z \in \mathit{Stream}_Z$, $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, we have the following calculation:

$$
\begin{array}{ll}
 & (\mathit{ref}_W \star \mathit{ref}_X).\Delta_0.y_0.z \wedge (g_1 \wedge g_2).\Delta.z \\
\Rightarrow & \quad \text{assumption } y_0 = w_0 \Cup x_0 \\
 & (\mathit{ref}_W.\Delta_0.w_0.z \star \mathit{ref}_X.\Delta_0.x_0.z) \wedge (g_1 \wedge g_2).\Delta.z \\
\Rightarrow & \quad \wedge \text{ distributes over } \star, \text{ logic} \\
 & (\mathit{ref}_W.\Delta_0.w_0.z \wedge g_1.\Delta.z) \star (\mathit{ref}_X.\Delta_0.x_0.z \wedge g_2.\Delta.z) \\
\Rightarrow & \quad \text{assumption } (g_1 \Vdash_{W,Z} \mathit{ref}_W) \wedge (g_2 \Vdash_{X,Z} \mathit{ref}_X) \\
 & (\exists w : \mathit{Stream}_W \bullet (w_0 \overset{\Delta_0}{=\!=} w) \wedge \mathit{ref}_W.\Delta.w.z) \star (\exists x : \mathit{Stream}_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge \mathit{ref}_X.\Delta.x.z)
\end{array}
$$

$=$ logic, assumption $W \cap X = \varnothing$

$\exists w{:}\,Stream_W, x{:}\,Stream_X \bullet (w_0 \Cup x_0 \overset{\Delta_0}{=\!=} w \Cup x) \wedge (ref_W.\Delta.w.z \star ref_X.\Delta.x.z)$

$=$ logic, assumption $y_0 = w_0 \Cup x_0$

$\exists w{:}\,Stream_W, x{:}\,Stream_X \bullet (y_0 \overset{\Delta_0}{=\!=} w \Cup x) \wedge (ref_W \star ref_X).\Delta.(w \Cup x).z$

$=$ $W \cup X = Y$ and $W \cap X = \varnothing$

$\exists y{:}\,Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge (ref_W \star ref_X).\Delta.y.z$ □